NumCSE exercise sheet 4
Convolution and FFT, filtering, Lagrange interpolation

alexander.dabrowski@sam.math.ethz.ch
oliver.rietmann@sam.math.ethz.ch

November 21, 2018

Exercise 4.1. *(Long exercise) Sparse vectors, their convolution and FFT*

Given a vector x of numbers, we want to represent it with a new data structure which should be suitable in case most of its entries are negligible. We also want to implement convolution and fast Fourier transform for this new data structure.

(a) Consider the following two structs:

- template<class T> duplet, which has members
  - int ind, the index of an element in x;
  - T val, the value of the element at position ind;
- template<class T> sparse_vec, which has members
  - double tol. We consider negligible any number with absolute value smaller than tol;
  - duplets, a vector of objects duplet<T> which stores the indices and values of the non-negligible elements of x;
  - len, an integer which records the length of x.

Implement in sparse_vec<T> the following utility methods:

- void append(int ind, T val), which appends to duplets a new duplet with index ind and value val, if abs(val)>=tol;
- void cleanup(), which sorts duplets with respect to ind, eliminates repetitions, with the convention that if two elements have the same index their values should be added up, and eliminates any element with norm smaller than tol or with index larger than len-1. If x.cleanup() leaves x as it is, we say that x is **clean**;
- T get_val(int ind), which returns the element of x in position ind. It should work in $O(\log$ duplets.size()$)$ time and assumes that x is clean.

Solution:

```
template<class T> struct duplet {
    int ind;
    T val;

    duplet(int p, T v) {
        ind = p;
        val = v;
    }
};
```

```
template<class T> struct sparse_vec {
    double tol = 1e-6;
    vector<duplet<T> > duplets;
    int len=0;

    sparse_vec(int l) {
        len = l;
    }

    void append(int ind, T val) {
        if (abs(val) >= tol) {
            duplet<T> newduplet(ind, val);
            duplets.push_back(newduplet);
        }
    }

    void cleanup() {
        std::sort(duplets.begin(), duplets.end(),
            [] (duplet<T> x, duplet<T> y) { return x.ind < y.ind; });

        vector<duplet<T> > newduplets;
        T tmp = 0;
        for (int i=0; i<duplets.size(); i++) {
            if (i == duplets.size()-1 || duplets[i+1].ind != duplets[i].ind) {
                tmp += duplets[i].val;
                if (abs(tmp) >= tol) {
                    duplet<T> newduplet(duplets[i].ind, tmp);
                    newduplets.push_back(newduplet);
                }
                tmp = 0;
            }
            else {
                tmp += duplets[i].val;
            }
        }
        duplets = newduplets;
    }

    T get_val(int ind) const {
        if (duplets.empty())
            return 0;
        return _get_val(ind, 0, duplets.size());
    }

    T _get_val(int ind, int n1, int n2) const {
        if (n2 < n1)
            return 0;
        int m = (n1 + n2)/2;
        if (duplets[m].ind == ind)
            return duplets[m].val;
        if (duplets[m].ind < ind)
            return _get_val(ind, m+1, n2);
        if (duplets[m].ind > ind)
            return _get_val(ind, n1, m-1);
```

```
    }
```

sparse_conv_fft.cpp

Example: A representation of $x = (0, 0, 1, 0, 7 + i, 0)$ can be obtained with a `sparse_vec` with `duplets` = {`(ind=4,val=6-I), (ind=1,val=1e-7), (ind=22,val=9), (ind=4,val=1+2I), (ind=2,val=1)`}, `len=6`, `tol=1e-6`. Notice that `duplets.size()!=len`. Running `cleanup()` on such an object, `duplets` would become {`(ind=2,val=1), (ind=4,val=7+I)`}.

Important notes: In all the subproblems below, you can assume all vectors of type `sparse_vec` which are passed as inputs are already clean. Do not convert the `sparse_vec`s to/from dense vectors.

(b) In `struct sparse_vec` implement a method `cwise_mult` which returns the component-wise multiplication between `sparse_vec a` and `sparse_vec b` in $O(sz_a + sz_b)$ time, where sz_a, sz_b are respectively the number of non-negligible elements of `a` and `b`.

Solution:

```
    static sparse_vec cwise_mult(const sparse_vec &a, const sparse_vec &b) {
        sparse_vec out(max(a.len,b.len));
        int ia = 0, ib = 0;
        while(ia < a.duplets.size() && ib < b.duplets.size()) {
            int inda = a.duplets[ia].ind, indb = b.duplets[ib].ind;
            if (inda >= a.len || indb >= b.len) break;
            if (inda == indb) {
                out.append(inda, a.duplets[ia].val * b.duplets[ib].val);
                ia++;
                ib++;
            }
            else if (inda < indb)
                ia++;
            else if (inda > indb)
                ib++;
        }
        return out;
    }
```

sparse_conv_fft.cpp

(c) Implement a method `conv` which returns the discrete convolution of `sparse_vec a` and `sparse_vec b` in $O(sz_a\, sz_b)$ time.

Solution:

```
    static sparse_vec conv(const sparse_vec &a, const sparse_vec &b) {
        sparse_vec out(a.len + b.len - 1);
        for (auto x : a.duplets) {
            for (auto y : b.duplets) {
                out.append(x.ind + y.ind, x.val * y.val);
            }
        }
        return out;
    }
```

sparse_conv_fft.cpp

(d) Implement a method `fft` which returns the discrete Fourier transform of `sparse_vec x` in $O(n(\log n)^2)$ time, where n = `x.len`. You can assume that n is a power of 2. Your function should return a `sparse_vec` which is clean.

(*Optional*) Improve your code so that it runs in $O(n \log n)$. Even if you don't implement this improvement, you can suppose in the following subproblems that the runtime of `fft` is $O(n \log n)$.

Solution:

```
    static sparse_vec fft(const sparse_vec &x) {
        int n = x.len;
        if (n<=1) return x;

        sparse_vec even(n/2);
        sparse_vec odd(n/2);
        for (auto duplet : x.duplets) {
            if (duplet.ind % 2 == 0)
                even.append(duplet.ind/2, duplet.val);
            else
                odd.append((duplet.ind-1)/2, duplet.val);
        }

        sparse_vec f0 = fft(even);
        sparse_vec f1 = fft(odd);

        T omega = exp(-2.*PI/n*I);
        T s(1.,0.);
        sparse_vec tot(n);

        /* O(n*(log(n))^2) solution
        for (int k=0; k<n; k++) {
            tot.append(k, f0.get_val(k % (n/2)) + f1.get_val(k % (n/2))*s);
            s *= omega;
        }
        */

        /* Another O(n*(log(n))^2) solution
        for (auto p : f0.duplets) {
            tot.append(p.ind, p.val);
            tot.append(p.ind + n/2, p.val);
        }
        for (auto p : f1.duplets) {
            tot.append(p.ind, p.val * std::pow(omega, p.ind));
            tot.append(p.ind + n/2, p.val * std::pow(omega, p.ind + n/2));
        }

        tot.cleanup();
        */

        auto merge_insert = [&] (int shift) {
            int i0 = 0, i1 = 0;
            int f0_size = f0.duplets.size(), f1_size = f1.duplets.size();
            while (i0 < f0_size && i1 < f1_size) {
                int ind0 = f0.duplets[i0].ind + shift;
                int ind1 = f1.duplets[i1].ind + shift;
                if (ind0 == ind1) {
                    T tmp = 0;
                    do {
                        tmp += f0.duplets[i0].val;
                        tmp += f1.duplets[i1].val * pow(omega, ind0);
                        i0++; i1++;
```

```
                    } while (i0 < f0_size && i1 < f1_size &&
                        f0.duplets[i0].ind == ind0 && f1.duplets[i1].ind == ind0);
                    tot.append(ind0, tmp);
                }
                else if (ind0 < ind1) {
                    tot.append(ind0, f0.duplets[i0].val);
                    i0++;
                }
                else if (ind0 > ind1) {
                    tot.append(ind1, f1.duplets[i1].val * pow(omega, ind1));
                    i1++;
                }
            }
            while(i0 < f0_size) {
                int ind0 = f0.duplets[i0].ind + shift;
                tot.append(ind0, f0.duplets[i0].val);
                i0++;
            }
            while(i1 < f1_size) {
                int ind1 = f1.duplets[i1].ind + shift;
                tot.append(ind1, f1.duplets[i1].val * pow(omega, ind1));
                i1++;
            }
        };
        merge_insert(0);
        merge_insert(n/2);

        return tot;
    }
```

sparse_conv_fft.cpp

(e) Implement a method `ifft` which returns the inverse discrete Fourier transform of `sparse_vec` `x` in $O(n \log n)$ time. You can assume that n is a power of 2.

Solution:

```
    static sparse_vec ifft(const sparse_vec &x) {
        double n = x.len;
        sparse_vec out(n);
        sparse_vec x_conj(n);
        for (auto duplet : x.duplets) {
            x_conj.append(duplet.ind, std::conj(duplet.val));
        }
        for (auto duplet : fft(x_conj).duplets) {
            out.append(duplet.ind, std::conj(duplet.val)/n);
        }
        return out;
    }
```

sparse_conv_fft.cpp

(f) Let `a` and `b` be two complex vectors of length $n+1$ and n respectively. Assume that n is a power of 2. Implement a method `conv_fft` which returns the discrete convolution of `a` and `b` in $O(n \log n)$ time.

Solution:

```
    static sparse_vec conv_fft(sparse_vec a, sparse_vec b) {
        int N = a.len + b.len - 1;
        a.len = N;
        b.len = N;
        return ifft(cwise_mult(fft(a), fft(b)));
    }
```

sparse_conv_fft.cpp

(g) Can `conv_fft(x, y)` be asymptotically slower than `conv(x, y)` for a particular choice of `x` and `y`? Briefly motivate your answer.

Solution: Yes, for instance if `x` and `y` are very sparse but their discrete Fourier transforms are not. For example if `x` and `y` have all zero elements except for a single non-negligible 1 in the first position, their `fft` are the constant 1 vectors. Then `conv_fft` must at least calculate all these ones, while `conv` will just multiply together two numbers to get the result.

Exercise 4.2. *2D convolution, FFT2 and Laplace filter*

We review the two dimensional convolution, its relation with the discrete Fourier transform, and implement a discrete Laplacian filter.
Consider the 2-dimensional infinite arrays $X = (X_{k_1,k_2})_{k_1,k_2 \in \mathbb{Z}}$ and $Y = (Y_{k_1,k_2})_{k_1,k_2 \in \mathbb{Z}}$. We can define their convolution as the 2-dimensional infinite array $X * Y$ such that

$$(X * Y)_{k_1,k_2} = \sum_{j_1=-\infty}^{\infty} \sum_{j_2=-\infty}^{\infty} X_{j_1,j_2} Y_{k_1-j_1,k_2-j_2}.$$

In the same way as in the 1-dimensional case, given two matrices $X \in \mathbb{R}^{m_1,m_2}$ and $Y \in \mathbb{R}^{n_1,n_2}$ we can define their *discrete convolution* as the convolution between the zero extensions of X and Y trimmed of unnecessary zeros, and their *circular convolution* as the smallest period of the convolution between the periodic extension of X and the zero extension of Y.
Consider two matrices $A \in \mathbb{R}^{n,m}$ and $F \in \mathbb{R}^{k,k}$ with $k < n, m$.

(a) How should we extend A and F to larger matrices $\tilde{A}$ and $\tilde{F}$ so that the discrete convolution of A with F is equal to the circular convolution of $\tilde{A}$ with $\tilde{F}$?

Solution: By padding A and F with zeros so that the dimensions of $\tilde{A}$ and $\tilde{F}$ are $L_r \times L_c$ where $L_r =$(number of rows of A) + (number of rows of F) $- 1$ and $L_c =$ (number of columns of A) + (number of columns of F) $- 1$.

(b) By recalling to the 1-dimensional fast fourier transform (see the `Eigen::FFT` module), implement a function `fft2` which returns the 2-dimensional fast fourier transform of a matrix.

Solution:

```
template <typename Scalar> void fft2(MatrixXcd &C, const MatrixBase<Scalar> &Y) {
    int m = Y.rows(), n = Y.cols();
    C.resize(m,n);
    MatrixXcd tmp(m,n);

    FFT<double> fft; // Helper class for DFT
    // Transform rows of matrix Y
    for (int k = 0; k < m; k++) {
        VectorXcd tv(Y.row(k));
        tmp.row(k) = fft.fwd(tv).transpose();
    }

    // Transform columns of temporary matrix
    for (int k = 0; k < n; k++) {
        VectorXcd tv(tmp.col(k));
        C.col(k) = fft.fwd(tv);
    }
}
```

fft2.cpp

(c) Implement a function `ifft2` which returns the 2-dimensional inverse fast fourier transform of a matrix.

Solution:

```
template <typename Scalar> void ifft2(MatrixXcd &C, const MatrixBase<Scalar> &Y) {
    int m = Y.rows(), n = Y.cols();
    fft2(C, Y.conjugate());
    C = C.conjugate()/(m*n);
```

```
}
```

fft2.cpp

(d) Implement an efficient function with arguments A and F which computes their discrete convolution.

Hint: use the result derived in the previous steps and the two dimensional circular convolution theorem.

Solution:

```
void conv2(MatrixXcd &C, const MatrixXcd &A1, const MatrixXcd &A2) {
    // returns discrete convolution between A1 and A2 (fft implementation)
    int n1 = A1.rows();
    int m1 = A1.cols();
    int n2 = A2.rows();
    int m2 = A2.cols();
    int Lrow = n1+n2-1;
    int Lcol = m1+m2-1;

    MatrixXcd A1_ext = MatrixXcd::Zero(Lrow, Lcol);
    MatrixXcd A2_ext = MatrixXcd::Zero(Lrow, Lcol);
    MatrixXcd tmp1 = MatrixXcd::Zero(Lrow, Lcol);
    MatrixXcd tmp2 = MatrixXcd::Zero(Lrow, Lcol);
    A1_ext.topLeftCorner(n1,m1) = A1;
    A2_ext.topLeftCorner(n2,m2) = A2;

    fft2(tmp1, A1_ext);
    fft2(tmp2, A2_ext);
    ifft2(C, tmp1.cwiseProduct(tmp2));
}
```

fft2.cpp

(e) The discrete Laplacian filter is defined as

$$F = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}.$$

It is often used to detect edges in images. Test your filter on the black and white "picture" provided in the template.

Exercise 4.3. *Lagrange interpolation.*

Fix $n \in \mathbb{N}_0$ and let $t_0, \dots, t_n \in \mathbb{R}$ be distinct nodes, i.e. $t_i \neq t_j$ if $i \neq j$. Then

$$L_i(t) := \prod_{\substack{j=0\\ j\neq i}}^{n} \frac{t - t_j}{t_i - t_j}$$

is called the i-th *Lagrange polynomial* for these given nodes. For $y_0, \dots, y_n \in \mathbb{R}$ we call

$$p(t) := \sum_{i=0}^{n} y_i L_i(t)$$

the *Lagrange interpolant* through $(t_i, y_i)_{i=0}^n$.

(a) Prove that

$$\sum_{i=0}^{n} L_i(t) = 1$$

for all $t \in \mathbb{R}$.

Hint: Choose $y_i = 1$ for all $i \in \{0, \dots, n\}$ and use uniqueness of Lagrange interpolants.

Solution: Consider the interpolation points $(t_i, 1)_{i=0}^n$. Clearly, an interpolating polynomial for these points is $p \equiv 1$. By uniqueness of Lagrange interpolants, we have

$$1 = p(t) = \sum_{i=0}^{n} L_i(t)$$

for all $t \in \mathbb{R}$.

(b) For $t \in \mathbb{R}$ define $\omega(t) := \prod_{j=0}^{n}(t - t_j)$ and fix $i \in \{0, \dots, n\}$. Prove that $\omega'(t_i) \neq 0$ and

$$L_i(t) = \omega(t)\, \frac{\lambda_i}{t - t_i}$$

for all $t \in \mathbb{R}$, where

$$\lambda_i := \frac{1}{\omega'(t_i)}.$$

Solution: Fix $i \in \{0, \dots, n\}$. By the product rule, we have

$$\omega'(t) = \frac{\mathrm{d}}{\mathrm{d}t}(t - t_i) \prod_{\substack{j=0\\ j\neq i}}^{n}(t - t_j) = \prod_{\substack{j=0\\ j\neq i}}^{n}(t - t_j) + (t - t_i)\frac{\mathrm{d}}{\mathrm{d}t}\prod_{\substack{j=0\\ j\neq i}}^{n}(t - t_j)$$

for all $t \in \mathbb{R}$. In particular for $t = t_i$, we obtain

$$\omega'(t_i) = \prod_{\substack{j=0\\ j\neq i}}^{n}(t_i - t_j) \neq 0$$

since the nodes are distinct. Therefore,

$$L_i(t) = \frac{\prod\limits_{\substack{j=0\\ j\neq i}}^{n}(t - t_j)}{\prod\limits_{\substack{j=0\\ j\neq i}}^{n}(t_i - t_j)} = \frac{\prod\limits_{j=0}^{n}(t - t_j)}{(t - t_i)\omega'(t_i)} = \omega(t)\, \frac{\lambda_i}{t - t_i}$$

for all $t \in \mathbb{R}$.

i	t_i	y_i
0	-1	2
1	0	-4
2	1	6

Table 1: data

(c) Compute the Lagrange interpolant $p(t)$ corresponding to the data given in Table 1.

Solution: The Lagrange polynomials are given by

$$\begin{aligned} L_0(t) &= \frac{t^2 - t}{2} \\ L_1(t) &= 1 - t^2 \\ L_2(t) &= \frac{t^2 + t}{2} \end{aligned}$$

yielding the interpolant

$$p(t) = t^2 - t + 4(t^2 - 1) + 3(t^2 + t) = 8t^2 + 2t - 4.$$

(d) Use the Newton basis approach to compute the interpolating polynomial $\tilde{p}(t)$ for the data in Table 1.

Solution: We have

$$\tilde{p}(t) = \sum_{i=0}^{2} a_i N_i(t)$$

for all $t \in \mathbb{R}$, where

$$\begin{aligned} N_0(t) &= 1 \\ N_1(t) &= (t - t_0) \\ N_2(t) &= (t - t_0)(t - t_1) \end{aligned}$$

and

$$\begin{aligned} a_0 &= 2 \\ a_1 &= -6 \\ a_2 &= 8. \end{aligned}$$

Hence we obtain

$$\tilde{p}(t) = 2 - 6(t + 1) + 8t(t + 1) = 8t^2 + 2t - 4.$$

(e) Is $\tilde{p}(t)$ different from $p(t)$? Explain your answer.

Solution: We have $\tilde{p}(t) = p(t)$ for all $t \in \mathbb{R}$ because the interpolating polynomial is unique.

(f) What are the advantages of using the Newton basis compared to the Lagrange polynomials?

Solution: The Lagrange interpolant might become numerically unstable if some of the nodes t_i are close to each other, as it involves computing $t_i - t_j$ for $i \neq j$. In the Newton basis, we only compute $t - t_j$, so it is numerically more stable. Moreover, inserting a new node does not affect the previous coefficients in the Newton basis (while it does in the Lagrange basis).